N87-26565

MPP PARALLEL FORTH

John E. Dorband
Image Analysis Facility/Code 635
NASA/Goddard Space Flight Center
Greenbelt, MD 20771

ABSTRACT

MPP Parallel FORTH is a derivative of FORTH-83 and Unified Software Systems' Uni-FORTH. We will describe in this presentation the extension of FORTH into the realm of parallel processing on the MPP. With few exceptions, Parallel FORTH was made to follow the description of Uni-FORTH as closely as possible. Likewise, the parallel FORTH extensions were designed as philosophically similar to serial FORTH as possible. The Massively Parallel Processor (MPP) hardware characteristics, as viewed by the FORTH programmer, will be discussed. Then a description will be presented of how parallel FORTH is implemented on the MPP.

Keywords: FORTH, parallel languages, SIMD, MPP.

INTRODUCTION

The MPP is primarily capable of two types of processing, serial or scalar processing, and parallel processing. The MPP contains an array of 16,384 processing elements(PE's), the array unit. They are all given the same instruction at the same time; thus computing in this array can be viewed as serial processing on a single processor. Yet, processing is actually happening on all 16,384 processors at the same time. Each processor is a bit serial processor with 1024 bits of memory. Thus, the entire array contains 2 million bytes of memory and can be viewed as 1024 bit planes of 128x128 bits each.

The MPP array is a square mesh of processors, where each processor can pass data to its four adjacent processors. The edges of the mesh can be connected in various ways to form several other topologies. These topologies consist of such arrangements as a simple square, a torus, cylinders, or a helix. This communication arrangement allows the programmer to move, simultaneously, as much as 16,384 bits of data 64 processors away from their original source processor. These moves may be in one of four directions - north, west, south, and east.

As well as having a main control unit for scalar processing and an array unit for processing 16,384 elements of data in parallel, the MPP has a staging memory. This memory is the means by which data is moved from the host computer (VAX-11/780) into the array unit memory. The staging memory contains 32M bytes of memory, allowing it to be configured as 16,384 bit planes of 128x128 bits.

Therefore, the MPP, as viewed by the FORTH user, consists of essentially three main components: the main control unit (MCU) (the scalar processor and controller of the array), the array unit (ARU) (for parallel processing of data), and the staging memory (STG) (primarily used for I/O and as a large external bit plane memory).

If every processor had to perform every instruction given to it, it would be of little use as a general purpose computer. Conditional processing alleviates this problem. Conditional processing (such as the execution of an 'IF ... ELSE ... THEN' statement) on the array divides the processor into two groups of processors - those processors for which the condition is true and those for which the condition is false. Since processors can be individually told not to execute the current instruction, the processors for which the condition is true will only execute those instructions between the IF and the ELSE and those processors for which the condition is false will only execute those instructions between the ELSE and the THEN. Thus, through prudent use of conditional statements, the processors can be programmed to perform a range of different functions within the same general time span.

Parallel FORTH is implemented as simply and as straightforward as possible. A Uni-FORTH system is implemented on the MCU. Parallel extensions have been added to the kernel under a new vocabulary called PARALLEL. Context switching has been simplified so that the FORTH word '{' switches to the parallel vocabulary and '}' switches back to the vocabulary that was in use before the switch to the parallel vocabulary. This allows the user to redefine serial words as analogous parallel words under a parallel context, thus making it easier for the FORTH user to remember the new parallel words. For example, '+' normally means to add two numbers that are on the data stack, but in the parallel context, '{ + }', means add two 128x128 arrays of numbers on the array stack, which is in the ARU memory.

Two new stacks have been added to parallel FORTH that are not in serial versions of FORTH. These two stacks are the array stack (A) and the mask stack (M). The mask stack is not normally used or seen by the FORTH programmer. It is used to facilitate nested conditional statements, such as 'IF ... ELSE ... THEN' or 'BEGIN ... UNTIL'. The array stack is extensively used by the FORTH programmer, since it is the parallel equivalent of the MCU's data stack. Most operations that can be performed on elements of the data stack have corresponding operations that can be performed in parallel on the array stack, such as +, *, DUP, DROP, and ROT. There are a few other operations that are peculiar to the array stack.

The following sections will discuss in more depth the parallel operations that have been implemented to extend FORTH into the realm of parallelism.

VOCABULARY AND DATA DEFINITION

In MPP Parallel Forth there is a vocabulary called PARALLEL. All new parallel words are in this vocabulary except PE control unit (PECU) primitive words and mask stack operations. As pointed out in the introduction '{' and '}' are used to enter and exit the parallel vocabulary. The following is a definition that will manipulate the MCU data stack:

```
: MULTADD  * + ;
```

While the next definition manipulates the ARU array stack:

```
: MULTADD { * + } ;
```

Parallel variables can be allocated in either the array or the staging memory. If a user wants to allocate a 128x128 array of 7-bit values named AR1 in the staging memory, the following is used:

```
7 STG VARIABLE AR1
```

If a user wants to allocate a 128x128 array of 11-bit values in the array memory named AR2, the following is used:

```
11 ARU VARIABLE AR2
```

The definition of parallel constants is similar to defining variables, except the user puts an array on top of the array stack and then executes the statement:

```
13 ARU CONSTANT CON1
```

to create a 128x128 array of 13-bit constants. Likewise vectors and arrays of 128x128 arrays may be defined with VECTOR and MATRIX, respectively. A vector of 20 8-bit 128x128 arrays can be defined with the following statement:

```
20 8 ARU VECTOR VEC1
```

The word ALLOT will allocate variable space in either the stager or the array if it is preceded by the word STG or ARU, respectively. ALLOT is used by all the above-mentioned definition words.

PARALLEL I/O

Parallel files can be stored on the host in either matrix or image format. Each format allows for 8-bit, 16-bit and 32-bit values. A matrix format file contains multiple arrays of 128x128 values. An image format file contains multiple images of 512x512 values. The following command:

```
CHANA IMAGE8 OPEN WHAT.DAT
```

opens the file 'what.dat' as an image file of images with 8-bit values. LOAD is then used to read the matrix or image into a previously defined staging memory array. An image from an image file of 8-bit values should only be loaded into a VECTOR or MATRIX that has at least 8x16 or 128 bits allocated to it. The command to read a matrix into a stager array is:

```
V1 3 LOAD
```

This loads the third matrix of the current file into variable V1. To store an image into a file, the word STORE is used (i.e., V1 3 STORE).

MEMORY OPERATIONS

Memory operations are used to move data between the three MPP memories: the MCU, the ARU, and the staging memory. The word '@' fetches arrays from array variables in the stager and the ARU memory and puts them on the array stack. The word '!' stores an array from the array stack into an array variable in the stager or ARU memory. The word 'SCALAR' takes a value from the data stack in the MCU memory, broadcasts it to all PE's, and produces an array on top of the array stack that has the same value for all elements of the array. Also, when the context is the parallel vocabulary, any literals are compiled into constants that will be sent to the top of the array stack as a scalar value during execution. Operations such as GMAX, GMIN, and GOR can reduce an array of values into a scalar value that can be put onto the data stack.

ARRAY STACK OPERATIONS

Most array manipulation occurs on the array stack. The array stack consists of a stack of descriptors in MCU memory and the actual bit plane stack in the ARU memory. The array stack is manipulated by operations very similar to those used on the data stack. These operations consist of words such as DUP, DROP, SWAP, OVER, ROT, PICK, and ROLL. In addition to the standard stack operations there are also operations that are peculiar to the array stack. They consist of the following words: -NDROP, NDROP, A@, >A, ZERO, EXTRACT, SLIDE, EXG, CROSS, and TOPOLOGY.

NDROP drops the top n elements of the array stack. -NDROP skips the first n1 elements of the stack and drops the next n2 elements of the array stack. 'A@' copies the descriptor off of the MCU array stack onto the data stack. A parallel array descriptor consists of two values: the address of the least significant bit plane of the array(LSB) and the number of bit planes in the array(LEN). '>A' creates an array of n bit planes on the array stack, where n is taken from the top of the data stack. ZERO is the same as '>A' except the bit planes are initialized to zero. EXTRACT extracts a field of bits from the second element of the array stack and inserts it into a field of the same size in the top element of the stack. SLIDE slides the top element of the array stack across the array of PE's. EXG exchanges data in the top elements of the array stack among PE's of the ARU. CROSS exchanges data from the top elements of the array stack with the second elements of the array in different PE's of the ARU. The TOPOLOGY operation changes the topology of the ARU.

ARITHMETIC, LOGIC, AND COMPARISON OPERATIONS

All the operations in this section deal primarily with the elements on the top of the array stack. Basically they are analogous to the corresponding operations that operate on the top of the data stack. The difference is that operations on the array stack perform 16,384 operations at the same time instead of one at a time and values on the array stack can have variable numbers of bits instead of a fixed number such as 8, 16, or 32.

Normally operations on the data stack are either single or double precision. On the array stack, however, operations are classified as either fixed or variable precision. A fixed precision operation requires that both operands have the same length and that their result is the same length. A variable precision operation may operate on operands whose lengths are different. The result of such operations has a length that is dependent on both the specific operation and the length of the operands. All basic operations discussed here have a fixed precision operation. Only a few operations have both a fixed and a variable precision form of operation. These operations are +, -, *, /, MOD, and /MOD. Their variable precision forms are ~+, ~-, ~*, ~/, ~MOD, and ~/MOD.

The result of a ~+ or a ~- operation has a length equal to one plus the maximum of the two operands. The result of a ~* operation has a length equal to the sum of the length of the two operands. The length of result of a ~/ operation is the difference between the length of the dividend operand and the length of the divisor operand. Note that the length of the dividend must be larger than that of the divisor. The result of the ~MOD operation has a length equal to the length of the divisor operand. Since the result of the ~/MOD operation is the result of the ~/ operation followed by the ~MOD operation, the lengths of the results are the same as described for ~/ and ~MOD.

The fixed precision only operations are MAX, MIN, ABS, NEGATE, 1+, 1-, 2/, 2*, AND, OR, XOR, and NOT. Three special operations find the aggregate result and place it on the data stack. These global operations are global maximum(GMAX), global minimum(GMIN), and global or(GOR).

Comparison operations differ slightly from the other operations in this section in that they result in a value of length 1. These operations are <, =, >, 0<, 0=, and 0>.

CONTROL OPERATIONS

Control operations cause certain portions of code to be executed on some data and not on others. Parallel control is quite different from serial control. In serial control, condition evaluation determines whether or not a certain piece of code will be executed. In parallel control, both the code corresponding to the true condition and the false condition may have to be executed. Some of the processors must be turned off during the execution of the code for the true condition, then turned on for the execution of the code for the false condition. This is accomplished with a mask bit. It is set to one in processors whose data satisfy the condition, and to zero in those whose data does not. Thus only those processors that satisfy the condition execute the code for the true condition. The mask bit is then complemented and only those processors that did not satisfy the condition will execute the code for the false condition. As with execution of serial conditions, parallel conditions can be nested. Therefore, there is a mask stack. Mask stack primitive operations are used to implement the operations in this section.

The basic conditional structure is the IF ... ELSE ... THEN statement. The IF word duplicates the top element on the mask stack, takes the least significant bit of the top element of the array stack, and ands it to the top element on the mask stack. The ELSE word complements the top element of the mask stack. And the THEN word drops the top element of the mask stack.

The parallel conditional loop structure is also somewhat unusual. It continues to execute as long as there is a processor that has not met the condition to terminate the loop. The two types of loops are the BEGIN... UNTIL and the BEGIN ... WHILE ... REPEAT. The BEGIN word duplicates the top element of the mask stack. The REPEAT word marks the end of the loop. The WHILE word ands the least significant bit of the top element of the array stack to the top element of the mask stack and terminates the loop if no processor has the top element of the mask stack equal to one. The UNTIL word is the same as WHILE except the least significant bit of the top element of the array stack is complemented before it is anded to the top element of the mask stack.

Note that only certain operations are maskable. Therefore, one should be aware that operations may execute when the processor was masked out because the operation was not maskable. Generally, only operations that do not change the number of elements on the array stack or the order of the elements on the array stack are maskable. Thus, most stack manipulation operations and two operand operations are not maskable. See the MPP Parallel FORTH Word Reference for more specific details.

PECU AND MASK STACK PRIMITIVES

The PECU and mask stack primitives are not meant to be used by the general FORTH programmer. They are used by the primary parallel FORTH words to initiate actions to be performed in the ARU. If it is necessary to use them, they are described in the MPP Parallel FORTH word reference.

MPP PARALLEL FORTH WORD REFERENCE

Context Changing Words

PARALLEL

'PARALLEL' is the vocabulary that contains all the words that act on data in the MPP array unit.

{

This changes the context of word searches to the 'PARALLEL' vocabulary.

}

This returns the context to that specified prior to the change to the 'PARALLEL' vocabulary.

Arithmetic Words

+ (A: a1(n) a2(n) -- A: a3(n))

This adds a1 and a2, which are the same size and produces a3, which is that size.

- (A: a1(n) a2(n) -- A: a3(n))

This subtracts a2 from a1, which are the same size and produces a3, which is that size.

* (A: a1(n) a2(n) -- A: a3(n))

This multiplies a1 by a2, which are the same size and produces a3, which is that size.

/ (A: a1(n) a2(n) -- A: a3(n))

This divides a1 by a2, which are the same size and produces a3, which is that size.

MOD (A: a1(n) a2(n) -- A: a3(n))

This divides a1 by a2, which are the same size and produces the remainder a3, which is that size.

/MOD (A: a1(n) a2(n) -- A: a3(n) a4(n))

This divides a1 by a2, which are the same size and produces a3, the remainder, and a4, the quotient, which are of that size.

MAX (A: a1(n) a2(n) -- A: a3(n)) {maskable}

This finds the maximum of a1 and a2, which are the same size and produces a3, which is that size.

GMAX (A: a1(n) -- S: m) {maskable}

This finds the global maximum of the a1 and places it on the data stack as a scalar value.

MIN (A: a1(n) a2(n) -- A: a3(n)) {maskable}

This finds the minimum of a1 and a2, which are the same size and produces a3, which is that size.

GMIN (A: a1(n) -- S: m) {maskable}

This finds the global minimum of the a1 and places it on the data stack as a scalar value.

ABS (A: a(n) -- A: a(n)) {maskable}

This finds the absolute value of 'a' and replaces 'a' on the stack.

NEGATE (A: a(n) -- A: a(n)) {maskable}

This finds the 2's complement of the value of 'a' and replaces 'a' on the stack.

1+ (A: a(n) -- A: a(n)) {maskable}

This increments the value of 'a' and places it back on the stack.

1- (A: a(n) -- A: a(n)) {maskable}

This decrements the value of 'a' and places it back on the stack.

2/ (A: a(n) -- A: a(n)) {maskable}

This shifts a(n) to the right.

2* (A: a(n) -- A: a(n)) {maskable}

This shifts a(n) to the left.

~+ (A: a1(n) a2(m) -- A: a3(max(n,m)+1))

This adds a1 to a2 and produces a result, a3, which has a size that is the maximum of the sizes of a1 and a2, plus 1.

~- (A: a1(n) a2(m) -- A: a3(max(n,m)+1))

This subtracts a1 from a2 and produces a result, a3, which has a size that is the maximum of the sizes of a1 and a2, plus 1.

~* (A: a1(n) a2(m) -- A: a3(n+m))

This multiplies a1 by a2 and produces a result, a3, which has a size that is the sum of the sizes of a1 and a2.

~/ (A: a1(n) a2(m) -- A: a3(n-m))

This divides a1 by a2 and produces a result, a3, which has a size that is the difference of the sizes of a1 and a2.

~MOD (A: a1(n) a2(m) -- A: a3(m))

This divides a1 by a2 and produces the remainder, a3, which has a size of a2.

~/MOD (A: a1(n) a2(m) -- A: a3(n-m) a4(m))

This divides a1 by a2 and produces a remainder, a3, which has a size the same as a2 and a quotient, a4,which has a size that is the difference of the sizes of a1 and a2.

Logical Words

AND (A: a1(n) a2(n) -- A: a3(n))

This ands a1 and a2, which are the same size and produces a3, which is that size.

OR (A: a1(n) a2(n) -- A: a3(n))

This ors a1 and a2, which are the same size and produces a3, which is that size.

GOR (A: a1(n) -- S: m) {maskable}

This finds the global 'or' of the a1 and places it on the data stack as a scalar value.

XOR (A: a1(n) a2(n) -- A: a3(n))

This xors a1 and a2, which are the same size and produces a3, which is that size.

NOT (A: a(n) -- A: a(n)) {maskable}

This finds the complement value of 'a' and places it back on the stack.

Comparison Words

< (A: a1(n) a2(n) -- A: a3(1))

This determines if a1 is less than a2, and produces a bit plane that is 1 where it is true and 0 where it is false.

= (A: a1(n) a2(n) -- A: a3(1))

This determines if a1 is equal to a2, and produces a bit plane that is 1 where it is true and 0 where it is false.

> (A: a1(n) a2(n) -- A: a3(1))

This determines if a1 is greater than a2, and produces a bit plane that is 1 where it is true and 0 where it is false.

0< (A: a1(n) -- A: a2(1))

This determines if a1 is less than 0, and produces a bit plane that is 1 where it is true and 0 where it is false.

0= (A: a1(n) -- A: a2(1))

This determines if a1 is equal to 0, and produces a bit plane that is 1 where it is true and 0 where it is false.

0> (A: a1(n) -- A: a2(1))

This determines if a1 is greater than 0, and produces a bit plane that is 1 where it is true and 0 where it is false.

Stack Operation Words

DUP (A: a(n) -- a(n) a(n))

Duplicates the top element on the array stack.

DROP (A: a(n) --)

Drops the top element on the array stack.

NDROP (S: n A: a(m)...a(p) --- A: a(m) a(q))

Drops the top n elements of the array stack.

-NDROP (S: n1 n2 A: a1(p1)...a2(p2)...a3(p3)
--- A: a1(p1) a2(p2)...a3(p3))

Skips the first n1 elements of the array stack and drops the next n2 elements of the array stack.

SWAP (A: a1(n) a2(n) -- A: a2(n) a1(n))

Swaps the top two elements on the array stack.

OVER (A: a1(n) a2(n) -- A: a1(n) a2(n) a1(n))

Copies the second element on the array stack to the top of the stack.

ROT (A: a1(n) a2(n) a3(n) -- A: a2(n) a3(n) a1(n))

Moves a1 to the top of the array stack.

PICK (S: m A: a1(n) ... a2(n)
-- A: a1(n) ... a2(n) a1(n))

Copies the mth element of the stack to the top of the stack. (1 PICK is the same as OVER.)

ROLL (S: m A: a1(n) ... a2(n) a3(n)
-- A: ... a2(n) a3(n) a1(n))

Moves the mth element of the stack to the top of the stack. (3 ROLL is the same as ROT.)

DEPTH (-- S: n)

Returns the number of elements on the array stack.

A@ (A: a(n1) --- S: n2 n1)

Copies the first descriptor on the array stack onto the data stack.

>A (S: n --- A: a(n))

Creates an element on top of the array stack that has n bit planes.

ZERO (S: n --- A: a(n))

Create an element of size n that has a value of zero onto the top of the array stack.

EXTRACT (S: m1 m2 n A: a1(n1) a2(n2)
--- A: a1(n1) a2(n2))

Extracts a field of n bits of a1(n1) starting at m1 and places it in a2(n2) starting at m2.

SLIDE (S: n d A: a(p) -- A: a(p))

Slides 'a' n PE's in the direction designated by d. East if d=0, west if d=1, south if d=2, and north if d=3.

EXG (S: m1 m2 n d A: a(n) -- A: a(n))

Exchanges elements of 'a' n PE's apart in the direction designated by d. East/west if d=0, south/north if d=2. The addresses of mask bit planes are m1 and m2. The mask m1 determines which PE's accept data during the east or south portion of the move and m2 determines which PE's accept data after the west or north portion of the move.

CROSS (S: m1 m2 n d A: a1(n) a2(n)
-- A: a1(n) a2(n))

Exchanges elements of a1 with a2 n PE's apart in the direction designated by d. East/west if d=0, south/north if d=2. Elements of a1 move to the east or south and elements of a2 move to the west or north. The addresses of the mask bit planes are m1 and m2. The mask m1 determines which PE's accept data during the east or south portion of the move and m2 determines which PE's accept data after the west or north portion of the move.

TOPOLOGY (S: n ---)

The number n designates the topology of the array when an EXG, SLIDE, or CROSS is performed.

Topology	North/South Connection	East/West Connection
0	None	None
1	None	Cylinder
2	Cylinder	None
3	Cylinder	Cylinder
4	Open-spiral	None
5	Open-spiral	Cylinder
6	Closed-spiral	None
7	Closed-spiral	Cylinder

Memory Operation Words

@ (S: m --- A: a(n)) {maskable}

Moves an array variable described by a descriptor at address m onto the array stack.

! (S: n A: a(n) ---) {maskable}

Moves an array from the array stack into an array variable described by a descriptor at address m.

SCALAR (S: <scalar value> --- A: a(n))
{maskable}

Broadcasts a scalar value into array 'a' of all PE's.

LITERAL

Compiles a constant into a word that will be placed onto the array stack during execution, or will immediately place it on the stack during interpretation.

LIT

This is the execution time routine used to place a constant, compiled into the code, onto the array stack.

DESC (S: n --- S: n2 n1)

Fetches the descriptor at address n and places it on the data stack. The address of the least significant bit plane (LSB) of the variable is n2 and n1 is the size of the variable.

Control Words

IF (A: a(n) M: m1 --- M: m1 M: m2)

Creates a new layer on the mask stack that is the result of anding the least significant bit plane of 'a' and m1.

ELSE (M: m --- M: m)

Complements the value of the top element of the mask stack.

THEN (M: m ---)

Drops the top element of the mask stack.

BEGIN (M: m1 --- M: m1 m1)

Duplicates the top element of the mask stack.

WHILE (A: a(n) M: m1 --- M: m1)
or (A: a(n) M: m1 ---)

Ands the least significant element of 'a' and m1. If no element of m1 is equal to 1, the loop is terminated.

REPEAT

Marks the end of a BEGIN...WHILE...REPEAT loop.

UNTIL (A: a(n) M: m1 --- M: m1 M: m2)
or (A: a(n) M: m1 ---)

Ands the complement of the least significant element of 'a' and m1. If no element of m1 is equal to 1, the loop is terminated.

I/O Words

MATRIX8
MATRIX16
MATRIX32

File types for files that contain 128x128 arrays of 8, 16, or 32-bit values.

IMAGE8
IMAGE16
IMAGE32

File types for files that contain 512x512 images of 8, 16, or 32-bit values.

LOAD (S: n2 n1 ---)

Loads an array n1 or image n1 from the currently opened file into the designated bit plane described by the descriptor at address n2.

STORE (S: n2 n1 ---)

Stores the designated bit planes described by the descriptor at address n2 into an array n1 or image n1 of the currently opened file.

Defining Words

VARIABLE (S: n f ---)

Allocates an n bit plane variable array in either the stager or the array.

CONSTANT (S: n f A: a(n) ---)

Allocates an n bit plane constant array in either the stager or the array and loads it with the top element of the array stack.

VECTOR (S: m n f ---)

Allocates a vector of m n bit values in either the stager or the array.

MATRIX (S: m1 m2 n f ---)

Allocates an m1xm2 matrix of n bit values in either the stager or the array.

Compiler Words

ALLOT (n f ---)

Allocates n bit planes in either the array (ARU) or the stager (STG).

ARU

Indicates that the desired variable will be allocated in the array.

STG

Indicates that the desired variable will be allocated in the stager.

PECU Primitive Words

PECU (S: <PECU address> --)

The word 'PECU' takes an address off the MCU data stack and places it in register 'SPE', which starts the PECU at that address.

S>C (S: <64 bit scalar> <LSB of scalar> ---)

The word 'S>C' loads the LSB of a scalar into PE0. The 64-bit scalar value will be loaded into the common register from the data stack.

C>S (--- S: <64 bit scalar> <LSB of scalar>)

The word 'C>S' stores the 64-bit return register A value on the data stack followed by the value 64.

A>PE2 (A: <descriptor> ---)
A>PE4 (A: <descriptor> ---)
A>PE6 (A: <descriptor> ---)

Takes 2 descriptors from the array stack and places them into registers PE2-PE3, PE4-PE5, or PE6-PE7, respectively. Each descriptor consists of a 16-bit LSB and a 16-bit size.

S>PE2 (S: n1 n2 ---)
S>PE4 (S: n1 n2 ---)
S>PE6 (S: n1 n2 ---)

Takes 2 words from the data stack and places them into registers PE2-PE3, PE4-PE5, or PE6-PE7, respectively.

Mask Stack Operations

A>M (A: a(n) M: m1 --- M: m1)

Ands the least significant bit plane of a(n) to m1. The mask stack pointer is maintained in PE1.

M>A (M: m1 --- M: m1 A: a(1))

Copies the top bit plane of the mask stack onto the top of the array stack.

MDROP (M: m1 ---)

Drops a mask from the mask stack.

MDUP (M: m1 --- M: m1 m1)

Duplicates top element on mask stack.

ACKNOWLEDGEMENTS

I would like to express my appreciation to Dick Fahey, Arne Henden, Dan Klinglesmith, and Archie Warnock for their advice and consultation on the FORTH philosophy and to Lisa Hamet for her help in the coding and implementation of the Uni-FORTH kernel on the MPP.

REFERENCES

1. Brodie, Leo, "Starting FORTH," Prentice-Hall, Inc., Englewood Cliffs, NJ., 1981.

2. Brodie, Leo, "Thinking FORTH," Prentice-Hall, Inc., Englewood Cliffs, NJ., 1984.

3. Henden, Arne, "Uni-FORTH User's Guide," Unified Software Systems, Columbus OH, 1985.